A formal proof of a protocol for communications over faulty channels using the Larch Prover

Boutheina Chetali

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A formal proof of a protocol for communications over faulty channels using the Larch Prover

Boutheina Chetali

N° 2476

Janvier 1995

PROGRAMME 2

ISSN 0249-6399

A formal proof of a protocol for communications over faulty channels using the Larch Prover

Boutheina Chetali*

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet Eureca

Rapport de recherche n2476 — Janvier 1995 — 22 pages

Abstract: This paper describes, by means of an example, how one may mechanically verify concurrent programs on the automated theorem prover LP. It presents a fully computer checked proof of a protocol for communications over faulty channels. The chosen specification environment is UNITY, since the proposed model can be fruitfully applied to a wide variety of problems and modified or extended for special purposes. It provides a higher level of abstraction to express solutions to parallel programming problems. We investigate how the UNITY methodology can be mechanized in LP, and how we can use the theorem proving methodology to prove safety and liveness.

Key-words: formal verification, concurrent program, protocol, faulty channels, Larch prover

(Résumé : tsvp)

soumis

*CRIN-CNRS and INRIA-lorraine, University of Henri Poincaré, B.P. 239, 54506 Vandoeuvre-lès-Nancy,email: chetali@loria.fr

Unité de recherche INRIA Lorraine
Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : (33) 83 59 30 30 – Télécopie : (33) 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ
Téléphone : (33) 87 20 35 00 – Télécopie : (33) 87 76 39 77

Une preuve formelle d'un protocole de transmission sur des cannaux défectueux utilisant le Larch prouveur

Résumé : Cet article décrit à travers un exemple, l'automatisation de la vérification de programmes concurrents à l'aide du prouveur de théorème LP. Il présente une preuve formelle d'un protocole de transmission à travers des cannaux défectueux. L'environnement de spécification choisi est UNITY, un modèle pouvant être appliqué à une large variété de problèmes, modifié et étendu pour des applications particulières. Nous avons étudié la formalisation de la méthodologie d'UNITY à l'aide du prouveur de théorèmes LP, et comment utiliser ce prouveur pour la preuve de propriété de *sûreté* et de *vivacité*.

Mots-clé : vérification formelle, programme concurrent, protocole, cannaux défectueux, Larch prouveur

1 Introduction

Several approaches to program verification have been proposed and used in the literature. Among them, there are two principal views, one operational and the other syntactic. The first one consists of analysing the program in terms of its execution sequences, analysis which is often successful in the case of sequential programs but much less so in the case of concurrent programs. The syntactic approach is based on an axiomatic reasoning, where one needs a formalism to express the relevant properties of a program, an appropriate language to construct well-founded formulas and a proof system to construct proofs. The UNITY environment [CM88] provides those tools to design and prove concurrent programs.

Despite the relative simplicity of the proposed model, the formal verification of a program specified in UNITY requires a careful analysis and involves an enormous amount of clerical detail. Therefore, a theorem prover is required to provide a very high degree of confidence in the correctness of the verification. UNITY offers a logic and a notation in which abstract specification of the computation and its successive refinements are expressed, without any mention to execution sequences. This *static* view of the program recall the LARCH style of specification [GHG+93], which emphasizes brevity and clarity rather than executability.

In this paper, we describe a formal proof of a concurrent program specified in UNITY and we show how UNITY logic and methodology could be specified within a general-purpose theorem prover for first order logic like LP, in order to verify safety and liveness properties. For that, we present the fully computer checked proof of a program describing a protocol for communications over faulty channels. Our aim is to show that we can take advantage of both the power and the simplicity of UNITY and LP, and that a theorem prover can be actually used to detect flows in a program specification, even simple one presented in textbooks.

The protocol studied in this paper is taken from Chapter 17 of the textbook of Chandy and Misra [CM88]. Primarily, our study was just intended to check our implementation of UNITY; whether the logic supported by LP can be used to encode the logic of UNITY, and whether the style of LP specification could be used to specify a UNITY program. Despite its simplicity, it contains all the difficulties and problems of a real protocol and allows us to easily illustrate the solutions we propose. Even in such a basic protocol presented by experts in UNITY, we found bugs and flaws, essentially two in our experiments.

First for the safety properties, the full mechanization of the proof lead us, as often in such an experiment, to enter low-level detail of the specification. This necessary refinement forces us to address assertions whose proof is either understood or left to the reader, and as expected we found there underestimation of the actual difficulty of the proof. Precisely Chandy and Misra's proof is based on the invariance of three predicates I_1, I_2 and I_3 and the authors understood that the invariance of each predicate can be proved independently. The experiment with LP has shown that the invariance of I_1 requires this of I_3 and this of an additional predicate.

Second for the liveness properties, we highlighted the fact that the low-level steps *left to the reader* carry the deeper aspect of the protocol, e.g. the data communication over

the faulty channels. Moreover the experiment has taught us that the strategy aspect in developing a proof of liveness cannot at all be neglected.

The paper is organized as follows. We start by a brief description of the mechanized implementation of UNITY in LP. Then we describe the communication protocol provided by Chandy and Misra [CM88], and we show the main steps of the mechanization of the correctness proof. In particular, we explain the flows detected in the safety proofs, and we describes the strategy used to prove progress properties. Finally, we conclude with a discussion about this study and future work.

We suppose a little familiarity with the LP system (as few as possible) and we recommand [GG91] as an introduction. This experiment has been conduct with the release 2.4 of Lp.

2 About the encoding of UNITY proof system in LP

For our purpose, a UNITY program consists of three sections: a **declare** section that declares the variables used in the program, an **initially** section that describes the initial values of the variables, and an **assign** section that consists of a non-empty set of assignment statements. Execution of every UNITY statement (assignment) terminates in every program state and a program execution consists of an infinite number of steps in which each statement is executed infinitely often. A UNITY program terminates by reaching a *fixed point*, which is equivalent to termination in standard sequential-programming terminology. Correctness of concurrent programs is defined in terms of properties of execution sequences. There are two basic kinds of properties of concurrent programs: *Safety*, the property must always be true, and *Liveness*, the property must eventually be true [Lam77]. There are five relations on predicates in UNITY theory: *Unless*, *Stable*, *invariant*, *Ensures* and *leads_to*. The first three are used for stating safety properties whereas the last two are used to express *Progress* properties, a subset of liveness properties.

The reader can find the specifications of the UNITY logic and methodology in LP in [Che95], with the encoding of the syntax of UNITY, the wp-calculus and the syntax of first order predicates. In the following, we give only the definitions needed in this paper.

Definitions of the Unity predicates

Proofs in UNITY are based on assertions of type $\{P\}s\{Q\}$, equivalent to $P \Rightarrow wp(s, Q)$, where $wp(s, Q)$ is the weakest pre-condition for the post-condition Q [Dij76]. The wp-calculus for assignment statements is formalized using the function: wp:act,bexp→bexp. act is the sort of actions, bexp is the set of boolean expressions. As this set is not (and cannot be) an extension of bool (a basic type provided by LP, we had to define *imply, or, and, true, false* as $\models$,$\vee$ and $\wedge$ different from LP's built-in operators $\Rightarrow$ | & for implication, disjunction and conjunction. So as for nnot,\= which are different from LP operators not,=. We also define arithmetic expressions (exp), boolean expressions may contain arithmetic subexpressions

$((x + y) > 0) \wedge b$. We also define actions and list of actions (act, alist), pairs of identifiers and expressions (id, exp) or $(id, bexp)$ and finally sets of pairs (pset) that we think of as being unordered lists of pairs of identifiers and expressions. Here is the axiomatization of wp:

```
Set name wp
assert
    wp(assg(i1.ex),p) == sub_bexp(p(i1.ex),p)
    wp(cond_assg(i1.ex,b),p) == (b |⇛ wp(assg(i1.ex),p)) ∧ (nnot(b) |⇛ p)
    wp(mult_assg(pl),p) == sub_bexp(pl,p)
    wp(cond_mult_assg(pl,b),p) == (b |⇛ wp(mult_assg(pl),p))∧(nnot(b) |⇛ p)
```

The function sub_bexp(list,p), where list a list of pairs (id,exp), substitutes each occurrence of id by exp in the boolean expression p. Let us now give the definition of the temporal predicates.

```
Set name unless
assert
    unless(p, q, anil) == true
    unless(p, q, a) == (((p∧¬(q))|⇛ wp(a, p∨q))=T)
    unless(p, q, a@act_l) == unless(p, q, a) & unless(p, q, act_l)

Set name ensures
assert
    ensures(p, q, anil) == false
    ensures(p, q, pgm) == unless(p, q, pgm) & exists_act(p, q, pgm)

Set name exist_act
assert
    exists_act(p, q, anil) == false
    exists_act(p, q, a) == (((p ∧¬(q)) |⇛ wp(a, q))=T)
    exists_act(p, q, a@act_l) == exists_act(p, q, a) ∨ exists_act(p, q, act_l)

Set name leads_to
assert
    when ensures(p, q, pgm) yield leads_to(p, q, pgm)
    leads_to(p, r, pgm) ∧ leads_to(r, q, pgm)) ⇒ leads_to(p, q, pgm)
    leads_to(p, r, pgm) ∧ leads_to(q, c, pgm)) ⇒ leads_to(p ∨ q, r ∨ c, pgm)
    leads_to(p, q, pgm) ∧ leads_to(r, q, pgm)) ⇒ leads_to(p ∨ r, q, pgm)

Set name Invariant
assert
Inv(p, pgm, init_cond) == ((init_cond |⇛ p)=T) & unless(p,F,pgm)
```

a@act_l is the list of all the actions of the program. For an atomic statement a, unless(p,q,a) means that if p holds and q does not in a program state, then after executing a either q or p holds ; hence, by induction on the number of statement executions, p keeps holding as long as q does not hold. The function exists_act checks whether there is a in pgm such that $\{p \wedge \neg q\}a\{q\}$ is valid. For a given program pgm, ensures(p,q,pgm) implies that unless(p,q,pgm) holds , and if p holds at any point in the execution of the program then q

holds eventually. invariant(p,pgm,cond_init) states that if p holds at every initial state and p is stable, then p holds at every state during any execution of pgm.

In order to infer p *leads_to* q once we have p *ensures* q in the current system, we use what LP calls a **deduction** rule and which it notes when *hypotheses* yield *conclusions*: the *when* clause contains the hypotheses of the rule and the *yield* clause, the fact inferred when the hypotheses are true. In the definition of the predicate leads_to [CM88], the disjunction rule is really an infinite set of rules, one for each integer n. Indeed, all of these rules are consequences of special case of the rule for $n = 2$. Furthermore, we code the disjunction rule for $n = 2$, and a special case of the general disjunction theorem [CM88].

An Induction principle for leads_to

The following induction rule is crucial because it involves the fundamental predicate, *leads_to* which usually helps to specify progress properties of programs. In most of the proofs of concurrent programs, in order to prove a progress property, we have to exhibit something which "decreases", so we need a rule to specify that fact.

$$\textbf{Induction}: \quad \frac{\forall m : m \in W :: p \wedge M = m \;\; \textbf{leads\_to} \;\; (p \wedge M <_w m) \vee q}{p \;\; \textbf{leads\_to} \;\; q}$$

W is a set well-founded under the relation $<_w$ and M is a function (the *metric*) from program states to W. The hypothesis of this rule is that from any program state in which p holds, the program execution eventually reaches a state in which q holds, or it reaches a state in which p holds with a lower value of the metric M. Since the metric value cannot decrease forever, eventually a state is reached in which q holds.
In LP, we state this principle as:

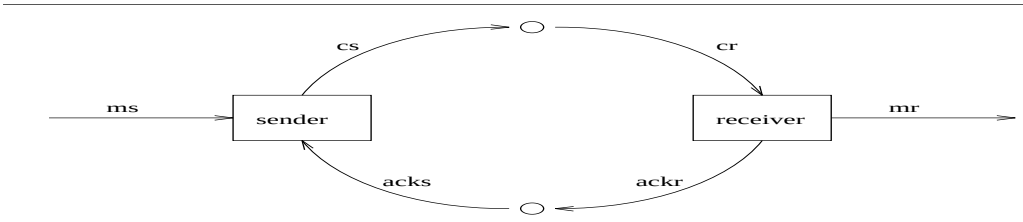
$$\textbf{IND}: \quad \frac{\mathsf{leads\_to}(\mathsf{p} \wedge \mathsf{eq}(\mathsf{l1}, \mathsf{l2}), (\mathsf{p} \wedge \mathsf{lexico}(\mathsf{l1}, \mathsf{l2})) \vee \mathsf{q}, \mathsf{pgm})}{\mathsf{leads\_to}(\mathsf{p}, \mathsf{q}, \mathsf{pgm})}$$

where eq is a function testing "equality" of two expression lists, and lexico formalize a lexicographic ordering on lists of arithmetic expression.

Note: Our implementation of UNITY contains all definitions, theorems, corollaries defined in [CM88]. Moreover, we do not code the axiom of substitution to preserve soundness [B.A91], which makes the proof system incomplete. All the theorems (derived rules) about *unless* and *ensures* are proved in our system [Che95], but not those about *leads_to*. Those proofs requires a structural induction on the length of proof that is not possible in LP.

3 A protocol of communication over faulty channels

In this section, we describes the fully computer checked proof of the protocol for communications over faulty channels proposed as a case studies by Chandy and Misra in [CM88]. The faulty channel and the protocol are specified as program specifications. Thus the task



of designing a communication protocol over faulty channels can be viewed as a program composition: the union of the protocol program and the faulty-channel program is a program which guarantees fault-free communication. In order to keep the length of the example within a reasonable size, only the basic ideas are sketched here.

A process called the *sender* has access to an infinite sequence of data, ms. Another process, the *receiver*, is required to output a sequence mr satisfying the following specification, where $|mr|$ is the length of mr:

invariant mr *is a prefix of* ms
$|mr| = n$ *leads_to* $|mr| = n + 1$

The sender and the receiver communicate through a faulty channel modeled by two pairs of nonfaulty channels (cs, cr) and $(ackr, acks)$, and a program that manipulates the contents of cs and cr. The sender adds messages to cs and receives acknowledgments from $acks$, the receiver receives message from cr and adds acknowledgments to $ackr$. The sender *writes* a data item and its index as a pair $(ks, ms[ks])$ in cs. So at each moment, ks represents the number of the last message sent by the sender. In the same way, the receiver receives from cr a pair (n, d). So it add the data item d to mr and sends n as an acknowledgment to the sender (using the channel $ackr$). At each time, kr is the number of the last message received by the receiver. Thus the sender sends a new message only upon receiving an acknowledgment for the previous one (see *Invariant* I_1 given below).

Specification of the faulty channel:

It is important to specify a faulty channel such that lost or duplication of messages are not under the control of the sender and the receiver, but may be done autonomously. **safety** properties state that a faulty channel can only remove the head of cs or append to the rear of cr; furthermore, a faulty channel may loose messages from cs and/or duplicate messages in cr. A **progress** specification of a faulty channel is that, if a message m is appended infinitely often to cs and no other message is appended to cs, then m appears in cr eventually. The corresponding specification of the faulty channel could be found in [CM88].

Chandy and Misra give the UNITY program, *FC*, which represents a faulty channel and the proof of its correctness. They state that this demonstration is not necessary for the study of communication protocols; only the specification of a faulty channel is required to develop

and verify protocols, thus we assume the correctness of the proof of the faulty channel and we take its specification as axioms in our proof. It is important to note that this work is not intented to be a proof of the adequacy formalization (modelization) of the communication protocol but the proof of the correctness of the protocol in the framework of UNITY. Readers who want to learn more about the formalization are referred to [CM88]. Indeed, we focus our study on the correctness proof of the program which specify the protocol and we show that the proposed proof is incomplete.

The protocol

A communication protocol is a program which fulfills the requirements of correct communication when it is composed with the program (FC) which meets the requirements of faulty channel.

Program {Protocol}
Declare $ks, kr : integer$
initially
$ks, kr = 1, 0 \ [] \ cs, cr, acks, ackr = null, null, null, null \ [] \ mr = null$
assign
$ackr := ackr; kr$
$[] \ ks, acks := ks + 1, tail(acks) \quad if \ acks \neq null \wedge ks = head(acks)$
$[] \ acks := tail(acks) \quad if \ acks \neq null \wedge ks \neq head(acks)$
$[] \ cs := cs; (ks, ms[ks])$
$[] \ kr, mr, cr := kr + 1, mr; head(cr).val, tail(cr) \quad if \ cr \neq null \wedge kr \neq head(cr).dex$
$[] \ cr := tail(cr) \quad if \ cr \neq null \wedge kr = head(cr).dex$
end{Protocol}

This program is slightly different from the one in [CM88], due to the not encoding of the parallel assignment ||. The two programs are equivalent under this modification.

Correctness of the protocol: to prove the correctness of the protocol, the following invariants have to be proved and the progress property given before.

Invariant I_1 : $\langle \forall y : y \in acks \vee y \in ackr :: y \leq kr \wedge ks \leq y + 1 \rangle \ \wedge$
$\langle \forall x : x \in cs \vee x \in cr :: kr \leq x.dex \leq ks \rangle \ \wedge$
$kr \leq ks \leq kr + 1$

Invariant I_2 : $\langle \forall x : x \in cs \vee x \in cr :: x.val \in ms \rangle \ \wedge$
$mr \ is \ a \ prefix \ of \ ms \ \wedge$
$|mr| = kr$

Invariant I_3 : $cs.dex, cr.dex, acks, ackr \ are$ sequences of nondecreasing integers in Protocol.

The **progress** condition is: $|mr|=n$ **leads_to** $|mr|=n+1$

4 Translation in LP

Much of the power of the deductive system of LP relies upon *normalization*, e.g. computation of normal forms by rewriting, then the basis for proofs is a logical system, which consists of a signature, equations, rewrites rules, operators theories,...etc. In addition, it provides inference rules and tactics for proving theorems. LP requires guidance in the proof strategy, e.g. the chaining of elementary proof tactics, and in the introductions of lemmas if necessary. These lemmas are proved and later used as axioms in proofs. In our proof, the suggestions we made were about the proof methods, or explicit instantiations of variables in lemmas, equations, ...etc. In the following, we explain the main steps in the translation of the UNITY program *protocol* in LP.

UNITY variables are of two kinds. The identifiers appear in the program text and are encoded as constant in LP of sort id. The proof variables appear in the UNITY proof and are encoded as LP variables of sort id_of_var. A UNITY program is defined as a list of actions and will be considered as a constant: prg : $\rightarrow$ actlist in LP.

Identifiers and naturals are themselves expressions. As the logic of LP does not has subsorts, we explicitly embedded id, id_of_var and nat into exp using functions id_to_exp and nat_to_exp. The distinction between expressions and basic sets yields the same distinction among variables, e.g. acks, ackr, cr, ... etc, are identifiers of sequence of sort id and appear in the program text, yvar and xrec are identifiers of variables of sort id_of_var needed in the proof obligations. To distinguish identifiers for substitutions, we define a particular identifier we call firstid, and an operator nextid:id $\rightarrow$ id. Therefore the sort id is generated by firstid and nextid. firstid is see to be the first identifier which appear in the text of the program, the second one is nextid(firstid) and so on.

A boolean expression init represents the **Initially** section of the program *protocol*:

```
assert
    init == (id_to_exp(kr) \= nat_to_exp(0))
            ∧ (id_to_exp(ks) \= nat_to_exp(succ(0)))
            ∧ (seq_to_exp(id_to_seq(ackr)) \= seq_to_exp(emptyseq))
            :
```

The **assign** section is encoded as a list of actions: proto == cons(a_1, cons(a_2, ..., cons(a_6, nil)))

As an example, the two actions:

$$\begin{array}{ll} ackr := ackr; kr & \\ [] \; ks, acks := ks+1, tail(acks) & if\ acks \neq null \wedge ks = head(acks) \end{array}$$

are translated in LP using the functions assg(id.exp) and cond_mult_assg((id.exp), bexp):

```
-------------------------------------------------------------------------------------
  cons(assg(ackr. seq_to_exp(id_to_seq(ackr) +- id_to_exp(kr))),
```

```
    cons(cond_mult_assg(p(ks. (id_to_exp(ks) + nat_to_exp(1)))
                        \union (p(acks.seq_to_exp(tail(id_to_seq(acks))))\union empty),
                        nnot(id_to_exp(acks)\=seq_to_exp(emptyseq))
                        ^(id_to_exp(ks) \= head(id_to_seq(acks)))),nil))
-------------------------------------------------------------------------------------
```

5 Proof obligations

5.1 Safety Properties

The proof obligations describe properties that the protocol must satisfy. They are the equational axiomatization of the three invariants I_1, I_2, I_3. For example, invariant I_1 states:

- For any acknowledgment y in the channel $ackr$ or $acks$, it must be lower than the index part of the last message received by the receiver ($y \leq kr$). Moreover y must be either equal to ks if the sender has not yet received an acknowledgment of value y or equal to $ks - 1$ (the sender has already received an acknowledgment of value y). This is translated by $ks \leq y + 1$. That means that when the receiver receives the $(n+1)^{th}$ message, the acknowledgment of the n^{th} message, may be already in the channel $ackr$ or $acks$, then $kr \geq n$. In the same way, when n is in $ackr$ or $acks$, the last number sent by the sender is equal to $n + 1$ (if the sender sent more than once the message n and has already received the acknowledgment), or to n if it has not yet received the acknowledgment n.
- For any message x (a data item and its index), sent along the channel represented by cs or cr, the index must be greater than the index of the last message received by the receiver (kr) and lower than the index of the last message sent by the sender (ks). Remember that the sender sends a new message only upon receiving an acknowledgment for the previous one. If not, it sends the same once more.
- The receiver receives only what has been sent ($kr \leq ks \leq kr + 1$).

$$\begin{array}{lll} \textbf{Invariant } I_1 : & \langle \forall y : y \in acks \vee y \in ackr :: y \leq kr \wedge ks \leq y+1 \rangle & \wedge \\ & \langle \forall x : x \in cs \vee x \in cr :: kr \leq x.dex \leq ks \rangle & \wedge \\ & kr \leq ks \leq kr+1 & \end{array}$$

To prove this invariant in our system, we must prove the following theorem in LP:

Inv(lb, proto, init) == true

The first line in the invariant I_1, $\langle \forall y : y \in acks \vee y \in ackr :: y \leq kr \wedge ks \leq y+1 \rangle$ is expressed in LP using the predicate inseq(e, seq) which checks if the element e belongs to the sequence seq. The universal quantification on y is implicitly asserted using a variable yvar of sort id_of_var :

$$[\mathsf{inseq(yvar, ackr)} \vee \mathsf{inseq(yvar, acks)}] \Rightarrow [(\mathsf{yvar} \leq \mathsf{kr}) \wedge (\mathsf{ks} \leq (\mathsf{yvar}+1))]$$

Let us describe the LP proof for the second action of the program proto. We show that LP allows us to construct proofs whose structure are identical to that of the hand proofs. Precisely, we explain the steps where hand proof use intuitive arguments, that are not accepted by the theorem prover.

Proof of invariant I_1:

(1): inv(Ib, cons(a_2, nil),init) == true

LP expands this equation using the definitions of invariant and unless:

(1) : ((init $\Rightarrow$ Ib) =T) & ((Ib $\Rightarrow$ wp(a_2, Ib))=T) == true

The command res by &-m directs LP to use the conjunction method. The goal is split in two subgoals:

(1.1) : ((init $\Rightarrow$ Ib)=T) == true
(1.2) : ((Ib $\Rightarrow$ wp(a_2, Ib))=T) == true

Let A, B, C be respectively the first, second and third subterms of the term Ib:
A: [inseq(yvar, ackr) $\vee$ inseq(yvar, acks)] $\Rightarrow$ [(yvar $\leq$ kr) $\wedge$ (ks $\leq$ (yvar +1))]
B: [inseq(xrec, cr) $\vee$ inseq(xrec, cs)] $\Rightarrow$ [(kr $\leq$ ind(xrec)) $\wedge$ (ind(xrec) $\leq$ ks)]
C: [(kr $\leq$ ks) $\wedge$ (ks $\leq$ kr+1)]

(1.1): ((init $\Rightarrow$ (A $\wedge$ B $\wedge$ C))=T)==true
(1.2): [(A $\wedge$ B $\wedge$ C) $\Rightarrow$ wp(cond_assg([(ks,ks+1),(acks,tail(acks))],
(ks \= head(acks) $\wedge$ nnot(acks \= \emptyseq))),
(A $\wedge$ B $\wedge$ C)) = T] == true

Proof of (1.1): Using the definition of init which represents the initial conditions:

```
init -> (kr \= 0)^(ks \= 1)^(acks \= emptyseq)^(ackr \= emptyseq)
        ^(cr \= emptyseq)^(cs \= emptyseq)^(mr \= emptyseq)
```

The goal is normalized (elimination of $\Rightarrow$):

(1.1): ([(kr $\models$ 0) $\wedge$ (ks $\models$ 1) $\wedge$ (acks $\models$ emptyseq) $\wedge$ (ackr $\models$ emptyseq)
$\wedge$ (cr $\models$ emptyseq) $\wedge$ (cs $\models$ emptyseq) $\wedge$ (mr $\models$ emptyseq)] = T)
$\Rightarrow$ ((A $\wedge$ B $\wedge$ C)=T) == true

res by $\Rightarrow$-m: LP generates the hypothesis:

(kr $\models$ 0) $\wedge$ (ks $\models$ 1) $\wedge$ (acks $\models$ emptyseq) $\wedge$ (ackr $\models$ emptyseq)
$\wedge$ (cr $\models$ emptyseq) $\wedge$ (cs $\models$ emptyseq) $\wedge$ (mr $\models$ emptyseq) == T

and the following subgoal:

(1.1.1): ((([inseq(yvar,emptyseq) ∨ inseq(yvar, ∅)] ⊨ [(yvar ≤ 0) ∧ (1 ≤ (yvar +1))])
∧ ([inseq(xrec, emptyseq) ∨ inseq(xrec, emptyseq)]
⊨ [(0 ≤ ind(xrec)) ∧ (ind(xrec) ≤ 1)])
∧ [(0 ≤ 1) ∧ (1 ≤ 0+1)]) = T) == true

: ((([F] ⊨ [(yvar ≤ 0) ∧ (1 ≤ (yvar +1))])
∧ ([F] ⊨ [(0 ≤ ind(xrec)) ∧ (ind(xrec) ≤ 1)])
∧ [(T) ∧ (T)])=T)
==true

: ((T ∧ T ∧ T)=T)==true

: (T=T)==true

: true ◇

Indeed, the goal (1.1.1) is normalized and divided in three subgoals, using the properties of ∧. Moreover, LP normalizes the goal using the rewrite rules of its current system. The connectives ∧, ∨, ⊨ are *eliminated*, replaced by the corresponding built-in operators of LP. It stops when the goal is in *normal form*.

Proof of (1.2):

(1.2): [((A ∧ B ∧ C) ⊨ wp(cond_assg((ks, acks:=ks+1, tail(acks)),
(ks \= head(acks) ∧ nnot(acks \= \emptyseq))),
(A ∧ B ∧ C)))] = T == true
: [((A ∧ B ∧ C) ⊨ [((ks \= head(acks) ∧ nnot(acks \= emptyseq)
⊨ sub_bexp(pl,(A ∧ B ∧ C)))
∧ (nnot(ks \= head(acks) ∧ nnot(acks \= emptyseq)
⊨ (A ∧ B ∧ C))])] = T == true

pl represents the list of pairs [(ks,ks+1),(acks,tail(acks))], corresponding to the action a_2. LP applies the definition of the substitutions, replacing ks by ks+1, acks by tail(acks) and then normalizes the goal using the definitions of ≤,+,...,etc:

: [((A ∧ B ∧ C) ⊨ [((ks \= head(acks) ∧ nnot(acks \= emptyseq))
⊨ ([inseq(yvar, ackr) ∨ inseq(yvar, tail(acks))]
⊨ [(yvar ≤ kr) ∧ ((ks+1) ≤ (yvar +1))]
∧ [inseq(xrec, cr) ∨ inseq(xrec, cs)]
⊨ [(kr ≤ ind(xrec)) ∧ (ind(xrec) ≤ (ks+1))]
∧ [(kr ≤ (ks+1)) ∧ ((ks+1) ≤ kr+1)]))
∧ (nnot(ks \= head(acks) ∧ acks ≠ ∅)
⊨ (A ∧ B ∧ C))])] = T == true

res by ⇒-m: LP adds the hypothesis ((A ∧ B ∧ C)=T) to its current list of facts in order to prove the second term. It transforms the hypotheses by replacing each variable by a constant, yvar by yvarc, xrec by xrecc. At the step (1.1), there was not such replacement because the hypothesis does not contain variables, but constants.

Hypotheses:
[inseq(yvarc, ackr) ∨ inseq(yvarc, acks)] ⊨ [(yvarc ≤ kr) ∧ (ks ≤ (yvarc +1))] == T
[inseq(xrecc, cr) ∨ inseq(xrecc, cs)] ⊨ [(kr ≤ ind(xrecc)) ∧ (ind(xrecc) ≤ ks)] == T
[(kr ≤ ks) ∧ (ks ≤ kr+1)] == T
Goals:

(1.2.1) : [((ks ⊭ head(acks) ∧ nnot(acks ⊭ emptyseq))
⊨ ([inseq(yvarc, ackr) ∨ inseq(yvarc, tail(acks))]
⊨ [(yvarc ≤ kr) ∧ (ks ≤ yvarc)]
∧ [inseq(xrecc, cr) ∨ inseq(xrecc, cs)]
⊨ [(kr ≤ ind(xrecc)) ∧ (ind(xrecc) ≤ (ks+1))]
∧ [(kr ≤ (ks+1)) ∧ (ks ≤ kr)]))] = T == true

(1.2.2) : [(nnot(ks ⊭ head(acks) ∧ nnot(acks ⊭ emptyseq)) ⊨ (A ∧ B ∧ C))] = T == true

Proof of (1.2.1):

(1.2.1) : [((ks ⊭ head(acks) ∧ nnot(acks ⊭ emptyseq))) = T]
⇒ [([inseq(yvarc, ackr) ∨ inseq(yvarc, tail(acks))]
⊨ [(yvarc ≤ kr) ∧ (ks ≤ yvarc)]
∧ ([inseq(xrecc, cr) ∨ inseq(xrecc, cs)]
⊨ [(kr ≤ ind(xrecc)) ∧ (ind(xrec) ≤ (ks+1))])
∧ [(kr ≤ (ks+1)) ∧ (ks ≤ kr)]) = T] == true

res by ⇒-m: LP generates a new hypothesis and normalizes the subgoal:

D : [((ks ⊭ head(acks) ∧ nnot(acks ⊭ emptyseq))) = T] == true

(1.2.1) : [([inseq(yvarc, ackr) ∨ inseq(yvarc, tail(acks))]
⊨ [(yvarc ≤ kr) ∧ (ks ≤ yvarc)]) = T
& ([inseq(xrecc, cr) ∨ inseq(xrecc, cs)]
⊨ [(kr ≤ ind(xrecc)) ∧ (ind(xrec) ≤ (ks+1))]) = T
& [(kr ≤ (ks+1)) ∧ (ks ≤ kr)]) = T] == true

res by &-m: LP splits the goal in four subgoals:

(1.2.1.1) : ([inseq(yvarc, ackr) ∨ inseq(yvarc, tail(acks))]
⊨ [(yvarc ≤ kr) ∧ (ks ≤ yvarc)]) = T == true

(1.2.1.2) : ([inseq(xrecc, cr) ∨ inseq(xrecc, cs)]
⊨ [(kr ≤ ind(xrecc)) ∧ (ind(xrec) ≤ (ks+1))]) = T == true

(1.2.1.3) : [(kr $\leq$ (ks+1))] = T == true

(1.2.1.4) : [(ks $\leq$ kr)] = T == true

and tries to prove each subgoal:

(1.2.1.1) : *see below*

(1.2.1.2) : (ind(xrec) $\leq$ ks) $\models$ (ind(xrec) $\leq$(ks+1)), using hyp B and the transitivity.

(1.2.1.3) : (kr $\leq$ ks) $\models$ (kr $\leq$(ks+1)), using hyp C and the transitivity

(1.2.1.4) : [((ks $\not\models$ head(acks) $\wedge$ nnot(acks $\not\models$ emptyseq))) = T] == true
: $\Rightarrow$ [(ks $\not\models$ head(acks))=T] & [nnot(acks $\not\models$ emptyseq) = T] ==true
: $\Rightarrow$ [(inseq(ks,acks))=T]==true
: [(ks $\leq$ kr) $\wedge$ (ks $\leq$ (ks +1))]=T==true using hypothesis A.
: [(kr $\leq$ ks)=T] & [(ks $\leq$ kr)=T]==true using hypothesis C
: [(kr $\not\models$ ks)=T]==true

To finish the proof, we must prove the subgoal (1.2.1.1), having hypotheses A,B, C, D. By cases, yvarc $\in$ ackr or yvarc $\in$ tail(acks), and using the hypothesis D, Lp infers:

[(yvarc $\leq$ kr) $\wedge$ (ks $\leq$ (yvarc + 1))]==true $\Rightarrow$ [(yvarc $\leq$ kr)]==true, [(ks $\leq$ (yvarc + 1))]==true

So the subgoal (1.2.1.1) is reduced to: [(ks $\leq$ yvarc)]==true

At this point, Lp tries all the hypotheses in its system and stops because it fails in deriving the desired fact. As we have seen, the mechanized proof is similar to a hand proof; to prove the three subgoals Lp uses the defined theory, which consists of the specification of the data types, the definitions of the temporal predicates, the definitions of $<$, $>$,...etc and the definitions of the arithmetic operators (+,$-$,...). Nevertheless, it cannot achieve the proof.

Let us now explain the "intuitive" proof: As explained in the beginning of this section, the first line of invariant 1 expresses the fact that acknowledgments (which are positive integers) in the channels *acks* or *ackr* are less than kr and greater than $ks - 1$. Let us take an illustrative example: consider the transmission of the $N + 1$ message, in two different cases. The first one without loss and the second one with a possibly loss of the message in the channels *cs* or *cr*. Assume that the first N messages are well transmitted, and we give only the state variables which are relevant for the example. In the following, we write *the message N* instead of *the message with an index part equal to N*. Remember that a message m sent along the channels *cs* is of the form $\langle d, n\rangle$, where $d = ms[n]$.

Figure 1 gives a schematic view of what happens in the channels *acks* and *acks* in the transmission without loss. We start in a state where:

- $ks = N$: The sender sent the message N.

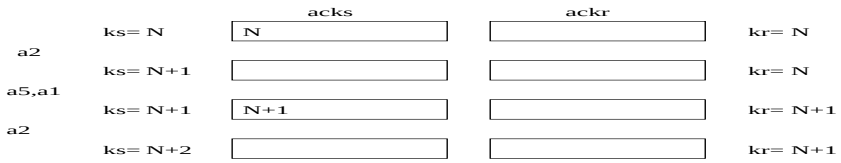


Figure 1: Transmission without loss.

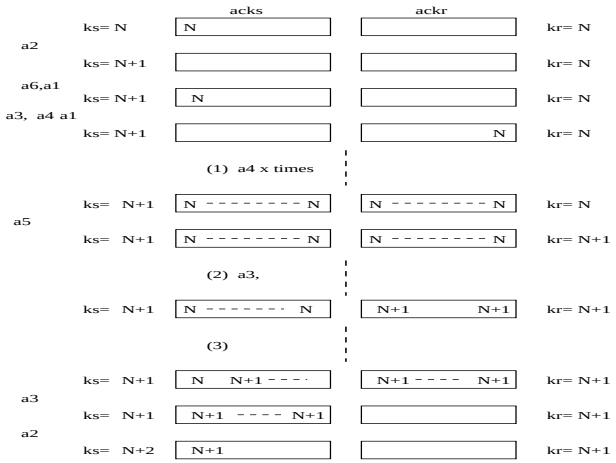


Figure 2: Transmission with the loss of the message N+1.

- $kr = N$: The receiver has received the message N and has sent the corresponding acknowledgment ($acks =< ..N.. >$).
- $head(acks) = N$: The acknowledgment gets to the head of $acks$.

The next state results from the execution of the second action a_2 of the program. The sender has compared the value of its counter, ks, with the value of the $head(acks)$. As $ks = head(acks)$, the sender increments its value by 1, and sends the message $N + 1$. The third state corresponds to the success of the transmission: The execution trace is of the form $< a_4, a_5, a_1 >$. The message $N + 1$ has been received by the receiver, which sends the acknowledgment $N + 1$. This acknowledgment is already in the head of the channel $acks$. The last state results from the execution of the action a_2, because $head(acks) = ks$. Then the sender sends the message $N + 2$.

Figure 2 shows the successfully transmission of the $(N + 1)^{th}$ message, with a possibly loss in the channels cs, cr. The first two states are similar to those of a transmission without loss; The sender sends the message $N + 1$ and waits for the acknowledgment. We assume that the message $N + 1$ is lost in the channels cs, cr.

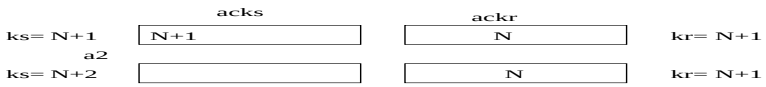


Figure 3: Unreachable state

- State 3: As the value of kr is not modified, the acknowledgment N is sent once time in the channels $ackr; acks$. The test $head(acks) = ks$ fails, ks is not modified. Then only the action $acks := tail(acks)$ is executed.
- State 4: The message $N + 1$ has been sent once more with the action a_4. We assume that the message $N + 1$ is lost x times.
- The state 5 resumes the following situation: x loss of the message $N + 1$ leads to at least x transmission of the acknowledgment N, so many occurrences of it appear in $ackr, acks$.
- State 6: The safety property of the faulty channels cs, cr ensures that only a finite number of the message $N + 1$ can be lost consecutively. Then state 6 corresponds to the reception by the receiver of the $(N + 1)^{th}$ message, $kr = N + 1$.
- The following states are for the acknowledgment of the message $N + 1$. Many occurrences of $N + 1$ appear in the channels $ackr$ corresponding to the execution of the first action. Eventually the acknowledgment gets to the head of $acks$. The test $head(acks) = ks$ succeed, then the sender sends the $(N + 2)^{th}$ message.

We note that the invariant 1 is verified:

- $\forall y \in acks \vee y \in ackr :: y \leq kr$. (State 6 : $y < kr$, State 7 : $y = kr$).
- $\forall y \in acks \vee y \in ackr :: ks - 1 \leq y$. (State 9 : $ks - 1 < y$, State 6 : $ks - 1 = y$).

We recall the subgoal in which we stop in the proof of invariant 1: $\mathsf{ks} \leq \mathsf{yvarc}$, with the hypotheses: $\mathsf{yvarc} \leq \mathsf{kr{=}{=}true}$, $\mathsf{ks} \leq (\mathsf{yvarc}+1) == \mathsf{true}$, $\mathsf{ks{=}head(acks){=}{=}true}$, $\mathsf{ks{=}kr{=}{=}true}$
Intuitively, we know that when $ks = kr$, all the acknowledgments are equal to ks or less than ks. In addition, if $head(acks) = ks$ then all the acknowledgments are equal to ks. (see figure2).

Now consider a state s_k in which $ks = kr$ and $(\forall y \in acks \vee y \in ackr :: (y \leq kr) \wedge (ks \leq (y+1)))$. Clearly invariant 1 holds in this state (Figure 3). As $ks = head(acks) \wedge not(acks = emptyseq)$, the action a_2 is enabled. We can execute a_2, and the resulting state s_{k+1} verify:
$ks = N + 2$, $kr = N + 1$, so $kr \leq ks \leq (kr + 1)$ and $\langle \exists y \in ackr :: not(ks \leq (y + 1)) \rangle$, with $y = N$.
Then invariant 1 does not hold in this state.

The problem comes from the fact that the state s_k is **unreachable**, with respect to the initial state. Thus this should not happen according to the program. Indeed we need

informations saying that acknowledgments in *acks* are smaller than those in *ackr*, and that *acks* and *ackr* are sequences of nondecreasing integers which is precisely Invariant 3. We need therefore an auxiliary invariant we call I_0:

Invariant I_0: $((\forall y \in acks) \wedge (\forall z \in ackr) :: y \leq z \wedge z \leq kr) \wedge (\forall y \in acks :: y \leq kr)$

This invariant must be proved before the other three. In fact, it restricts the all possible states of the program to the reachable states from the initial state. We prove the following theorem in LP:

Inv(((inseq(yvar, acks) $\wedge$ inseq(zvar, ackr)) $\models$ ((yvar $\leq$ zvar) $\wedge$ (zvar $\leq$ kr)))
$\wedge$ (inseq(yvar,acks) $\models$ (yvar $\leq$ kr)), proto,init)

Using invariant I_0 and invariant I_3 as axioms, we can now achieve the proof of the subgoal (1.2.1.1):

[(ks $\leq$ yvarc)]==true

Hypotheses:
yvarc $\leq$ kr==true
ks $\leq$ (yvarc +1) == true
ks=head(acks)==true
ks=kr==true
([inseq(yvarc, ackr) $\vee$ inseq(yvarc, tail(acks))]==true

Case 1: inseq(yvarc,tail(acks))==true
$\Rightarrow$ (head(acks) $\leq$ yvarc)==true , invariant 3.
$\Rightarrow$ (ks $\leq$ yvarc)==true

Case 2: inseq(yvarc,ackr)==true
$\Rightarrow$ (head(acks) $\leq$ yvarc)==true , invariant 0.
$\Rightarrow$ (ks $\leq$ yvarc)==true

end proof.

5.2 The proof of liveness

Proofs of liveness are somewhat different from those of safety. To prove *unless* or *ensures* properties, we apply the definition, it is not the case for *leads_to*. This is due to the fact that the first two have an "operational" definition and the last does not. When we deal with a property as p **leads_to** q, we have to check if it is "static" and then to try to prove p **ensures** q in order to infer p **leads_to** q. Otherwise, we must find "good intermediate fact" in order to use the other rules which specify *leads_to* as transitivity. To avoid this difficulty, we proceed by refinement or decomposition to simplify the conjecture. We exhibit

intermediate properties to be proved and rules to prove them. One of the most often used rule to prove *leads_to* properties is *cancellation rule*:

$$\textbf{CAN\_rule}: \quad \frac{leads\_to(p, q \vee b), leads\_to(b, r)}{leads\_to(p, q \vee r)}$$

After proving the three invariants given before, we prove the main property of the program *proto*, which is the specification of the program:

$$|mr| = n \quad leads\_to \quad |mr| = n + 1$$

The proof proposed by Chandy and Misra is the following, using kr for $|mr|$, from invariant (I2) ($\mapsto$ stands for leads_to):

$kr = n$	$\mapsto$	$kr = n + 1 \ \vee (n \in acks)$	(P_1)
$n \in acks$	$\mapsto$	$ks = n + 1$	(P_2)
$kr = n$	$\mapsto$	$(kr = n + 1) \vee (ks = n + 1)$	(P_3), CAN_rule on (P_1) and (P_2)
$ks = n + 1$	$\mapsto$	$kr = n + 1$	(P_4)
$kr = n$	$\mapsto$	$kr = n + 1$	CAN_rule on (P_3) and (P_4)

So we had to complete the proof given by the authors, and in particular we expand the proof of (P_2) and (P_4), as we are going to see.

Proof of (P_1): This property is proved using the *conditional property* (COND_prop) of a faulty channel [CM88]. In particular, we consider the faulty channel for the acknowledgments (transmission from the receiver to the sender), taking $p.m \equiv (kr = m)$.

Proof of (P_2): In order to prove this property, we first exhibit the fact that if an item n is in the sequence $acks$ then it will eventually be at the head of $acks$ or ks will be equal to $n + 1$ since the sender increments ks when he receives the acknowledgement for the n^{th} message. The problem is that we do not know *when*. Nevertheless, it is exactly the meaning of the induction principle (*see appendix*), using the position of the item in the sequence to be the metric: From any program state in which $(n \in acks)$ holds, the program execution eventually reaches a state in which $(n \in acks \wedge head(acks) = n) \vee ks = n + 1$, or it reaches a state in which $(n \in acks)$ holds and the position of n in the sequence $acks$ is lower (since the position of a item in a sequence cannot decrease indefinitely, $pos(head(seq)) = 1, pos(head(tail(seq))) = 2, \ldots ect$). On another hand, when an item n is at the head of $acks$ we know that ks will be equal to $n + 1$ (from the text of *proto*), we can use *ensures* to prove that. So the proof of $n \in acks \ \mapsto \ ks = n + 1$ is decomposed in:

$n \in acks$	$\mapsto$	$ks = n + 1 \vee (n \in acks \wedge head(acks) = n)$	IND
$n \in acks \wedge head(acks) = n$	$\mapsto$	$ks = n + 1$	
$n \in acks$	$\mapsto$	$ks = n + 1$	CAN_rule on the above two

Proof of (P_4): We have to prove the following property: $ks = n + 1 \ \mapsto \ kr = n + 1$, which asserts that when the sender sends the $(n + 1)^{th}$ message, the receiver eventually receives it. It is a progress property. The protocol is designed in such a way that if the message is

lost the first time it is sent, the sender sends it again unless it receives an acknowledgement. The safety property asserts that any message sent along the channel can be lost, but only a finite number of messages can be lost consecutively.

In order to prove (P_4), we have to decompose the problem: Fist of all, we want to use the conditional property of a faulty channel (COND_prop), in order to prove that if a message is in cs, it will eventually be in cr (it is the only way to prove passing of items between the sequences cs and cr). To do that, we take $p.m \equiv (ks = n+1 \wedge ((n+1, ms[n+1]) \in cs))$ and m as $(n+1, ms[n+1])$. When the message is in cr, intuitively we must prove that it will arrive to the head of cr, and then if the content of cr is m, kr will be equal to the index of m. As done for (P_2), we use the induction principle to prove that a message m in the tail of a sequence gets to the head.

Proof:

$ks = n+1 \;\mapsto\; p.m \vee kr = n+1$ (1)

$p.m \;\mapsto\; \neg(p.m) \vee (m \in cr)$ (2) COND_prop

$p.m$ **unless** $(m \in cr) \vee head(cr) = m \vee kr = n+1$ (3)

$p.m \;\mapsto\; (p.m \wedge m \in cr) \vee head(cr) = m \vee m \in cr \vee kr = n+1$ (4)

PSP_rule on (2) and (3).

$ks = n+1 \;\mapsto\; (p.m \wedge m \in cr) \vee head(cr) = m \vee m \in cr \vee kr = n+1$ (5)

CAN_rule on (1) and (4).

$p.m \wedge (m \in cr) \wedge pos(m, cr) = i$

$\mapsto\; (p.m \wedge (m \in cr) \wedge pos(m, cr) < i)$

$\vee (head(cr) = m \wedge m \in cr) \vee kr = n+1$ (6)

$p.m \wedge (m \in cr) \;\mapsto\; (head(cr) = m \wedge m \in cr) \vee kr = n+1$ (7)

(IND on the above)

$ks = n+1 \;\mapsto\; head(cr) = m \vee m \in cr \vee kr = n+1 \vee (m \in cr \wedge head(cr) = m)$ (8)

CAN_rule on the above and (5)

$head(cr) = m \;\mapsto\; m \in cr \wedge head(cr) = m$ (9) IMPL_leads

$ks = n+1 \;\mapsto\; kr = n+1 \vee m \in cr \vee (m \in cr \wedge head(cr) = m)$ (10)

CAN_rule on the above and (8)

$m \in cr \;\mapsto\; (m \in cr \wedge head(cr) = m)$ (11) IND

$ks = n+1 \;\mapsto\; kr = n+1 \vee (m \in cr \wedge head(cr) = m)$ (12)

CAN_rule on the above and (10)

$head(cr) = m \wedge m \in cr \;\mapsto\; kr = n+1$ (13)

$ks = n+1 \;\mapsto\; kr = n+1$

CAN_rule on the above and (13)

end proof.

We can represent this proof as a search tree, where the root is the predicate which holds in the starting state. Starting from a node, we expand the different possibilities and the goal of the proof is to reduce the tree to a single leaf, which represents the predicate that we want to obtain. The importance of the CAN_rule is that when we want to prove that $p \mapsto q$, we start from p and we try to understand what we want to obtain: whether we have a state where q holds or a state where r holds. So finding r is crucial, it represents a predicate hold in intermediate state *preceding* the state where q holds, and the CAN_rule allows to get rid of it.

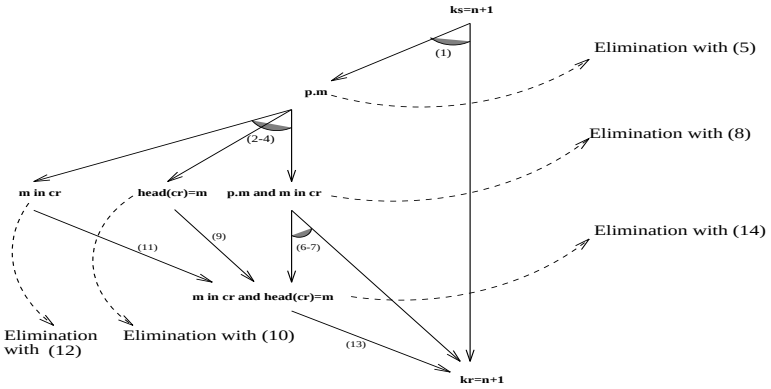


Figure 4: The application of the CAN_rule enables the elimination of the nodes, which represent the intermediate facts. Every triangle represents the application of CAN_rule, and closing the triangle means that the application succeed.

6 Discussion and Conclusion

In this paper, we have described, by means of an example, the mechanization of a UNITY proof using LP. The logic of the theorem prover in which it is intended to mechanize another notation must be both powerful and general enough to allow a sound and practical formulation. Several attempts to mechanize verification of UNITY programs through theorem provers have been undertaken, namely with NQTHM [Gol90], B_TOOLS [BM93], and HOL [And92].

Unlike the NQTHM system or B, the LARCH Prover does not contain any predefined theory. The design and development of the Larch proof assistant LP have been motivated primarily to debug LSL specification [GHG+93] but it has been also used to establish the correctness of hardware designs [SGG89], and to reason about algorithms involving concurrency [SGG88].
Our purpose in stating this work was to show the mechanization of an UNITY proof, to check whether the tactics provided by LP are sufficient to verify safety and liveness.

One might think that a theorem prover like B is well-suited for proving UNITY programs due to the fact that UNITY statements are assignments and that the *substitution* (substituting all occurrences of a variable by an expression) is a built-in mechanism in B. In the same way, we could use the NQTHM system which uses extensively predefined functions and which requires less guidance from the user whereas LP requires to reconstruct all basic functions. But this is an interesting feature since it gives the user more freedom for choosing the functions, for designing the proof and for debugging wrong specifications which is after all the main activity of the computer scientist faced to a computer proof of correctness.

Many researchers urge for more and more powerful provers and it may look paradoxical that we were happy with an assistant prover. The reason is that it helped us to find easily

bugs in wrong proofs and it allowed us to express properties we wanted to prove about individual programs. Indeed, we showed that LP allows the construction of a proof whose structure is identical to that of a careful hand proof. Nevertheless, low steps in hand proofs are usually achieved using intuitive arguments. As those arguments are often based on critical aspects of the problem, they must be formalized and mechanized in order to retrieve errors and mistakes.

The main contribution of this paper consists in taking advantage of both the simplicity and the power of the model proposed by Chandy and Misra and those of the assistant prover LP. Concurrent programs are difficult in essence. There is no hope to get them right without a formal specification of their behaviour and a careful proof of their correctness. We expect we have convinced the reader that we have reached the first aim through UNITY and the second through LP.

References

[And92] F. Andersen. *A theorem Prover for UNITY in Higher Order Logic.* PhD thesis, Technical University of Denmark, 1992.

[B.A91] Sanders B.A. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3:189–205, 1991.

[BM93] N. Brown and D. Mery. A proof environment for concurrent programs. In *In Proceedings FME93 Symposium*, volume 670 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[Che95] B. Chetali. Formal verification of concurrent programs: How to specify unity using the larch prover. Research Report 2475, Inria-Lorraine, 1995.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design*
A Foundation. Addison-Wesley, 1988.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[GG91] S. V. Garland and J. V. Guttag. A guide to lp, the larch prouver. Technical Report 82, Digital Ssystems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, USA., 1991.

[GHG+93] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[Gol90] D. M. Goldschlag. Mechanically verifying concurrent programs with the boyer-moore prover. *IEEE Transactions on Software Engineering*, 16(9):1005–1022, September 1990.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[SGG88] J. Staunstrup, S. J. Garland, and J. V. Guttag. Verification of VLSI circuits using LP. In *Proceedings of the IFIP WG 10.2 Conference on the Fusion of Hardware Design and Verification*, pages 329–345. Elsevier Science Publishers B. V. (North-Holland), 1988.

[SGG89] J. Staunstrup, S. J. Garland, and John V. Guttag. Localized verification of circuit descriptions. In J. Sifakis, editor, *Proceedings of a Workshop on Automatic Verification Methods for Finite State Systems, Grenoble (France)*, volume 407 of *Lecture Notes in Computer Science*, pages 349–364. Springer-Verlag, June 1989.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399